Formal methods in software development: A road less travelled

John A. van der Poll

School of Computing, UNISA, Pretoria, South Africa

ABSTRACT

An integration of traditional verification techniques and formal specifications in software engineering is presented. Advocates of such techniques claim that mathematical formalisms allow them to produce quality, verifiably correct, or at least highly dependable software and that the testing and maintenance phases are shortened. Critics on the other hand maintain that software formalisms are hard to master, tedious to use and not well suited for the fast turnaround times demanded by industry. In this paper some popular formalisms and the advantages of using these during the early phases of the software development life cycle are presented. Employing the Floyd-Hoare verification principles during the formal specification phase facilitates reasoning about the properties of a specification. Some observations that may help to alleviate the formal-methods controversy are established and a number of formal methods successes is presented. Possible conditions for an increased acceptance of formalisms in software development are discussed.

KEYWORDS: Automated reasoning, formal specification, heuristics, OTTER, Prover9, resolution, set theory, Vampire, verification, Z

1 INTRODUCTION

Formal methods in computing are considered to be the use of mathematical descriptions and techniques to develop (mainly) software systems through the use of formal *specifications*. This view stems from the 1980s and is prominent in amongst others, the book by Jim Woodcock and Martin Loomes [1]. However, *procedural* approaches to mathematical software development have their roots in the late 1960s already, owing to the work of Robert Floyd [2] and C.A.R. Hoare [3].

Of course, mathematical-based techniques in computing are much wider than just software development — automata theory [4] or relational database theory [5] are further examples of mathematical formalisms early on in computing. Nevertheless, the definition of 'formal methods' that is assumed in this paper is the one that embraces the use of mathematics in developing software.

Advocates of formal methods claim that such techniques facilitate the production of quality, verifiably correct, or at least highly dependable software and that the testing and maintenance phases are shortened. Critics on the other hand maintain that software formalisms are hard to master, tedious to use and not well suited for the fast turnaround times demanded by industry. Hence, the use of formalisms in software development has been a topic of a hot debate [6, 7].

In this paper some of these issues are addressed and it is argued that a formal-methods approach may be vital in the construction of correct software systems. A further contribution of this work lies in the integration of an abstract specification, constructed early on in the software development life cycle (SDLC) with traditional verification techniques normally applied during later phases.

The layout of the paper is: In Section 2, some of the early approaches to mathematical software development are presented and it is shown how the application of these could be crucial. Moving the mathematical rigour to an earlier phase in the SDLC presents some unique opportunities for identifying and removing errors early on in the design. To this end the idea of a formal specification phase embedding aspects of early mathematical approaches are presented in Section 3. An example of a Z specification is presented in Section 4 and the benefits to be gained in reasoning about the properties of a formal specification are considered.

Automated reasoning assistants may usefully be employed to aid the reasoning process and an early reasoner is discussed briefly in Section 5, fol-

Email: John A. van der Poll vdpolja@unisa.ac.za

lowed by the advances made by automated theorem provers in Section 6. The challenges brought about by set-theoretic reasoning are considered in Section 7 and it is shown that heuristics may further facilitate the reasoning process. Sections 8 and 9 address next-generation reasoners and notational complexities respectively. In Section 10 some formal-methods success stories are presented and we take a look in Section 11 at the conditions that formal methods in software engineering may have to meet in order to become a viable option in industry in future. Important observations are presented throughout. A summary concludes this paper.

2 FORMAL SOFTWARE DEVELOPMENT - PROCEDURAL APPROACHES

The work of Robert Floyd [2] and C.A.R. Hoare [3] together constitute some of the earliest writings on the formal verification of computer programs. Essentially Floyd defined rules for the construction of flowcharts while Hoare defined a number of similar deduction rules, based on the notions of preconditions, postconditions and statements for proving the correctness of programming constructs. These constructs were assignments, sequences of statements, conditionals (consequence) and looping constructs, all part of modern high-level languages like C or Java [8].

Examples of the verification of some of these constructs are presented next.

2.1 Hoare Logic: An Example

An expression of the form $\{P\}S\{Q\}$ where P and Q are properties of the program variables and S is a program (a single statement or large code fragment), is called a Hoare triple [3, 9]. P is called the precondition and Q is known as the postcondition of S. In this paper $\{P\}S\{Q\}$ is interpreted as follows: If statement S and all its associated variables are defined in context and precondition P is satisfied before the execution of S, execution of S is guaranteed to terminate, and afterwards, the program variables will satisfy Q [9]. This property is defined as *total correctness* by Baber [10]. Over time proof rules for the verification of assignment statements, the *skip* statement, conditionals, sequential composition, looping constructs, procedure calls, etc. were established. Details of some of these appear in Appendix B.

Verification theory involving pre- and postconditions allows us to solve a rather common problem in computing, namely, if the postcondition to a code fragment is known, how does one determine the precondition, i.e. where does one have to start to ensure the end result? The answer to this question is of value in requirements engineering [11] as well; if a user states some requirements (postcondition) and a software engineer suggests a solution, then what should be in place beforehand to ensure that the solution proposed will satisfy the requirements of the user, i.e. what is the precondition?

The following example illustrates an application of two of the Floyd-Hoare verification rules.

Example 1

Suppose one has to calculate the precondition, P for the sequence of statements $x := x-1$ and $y := y-1$, given a postcondition $\{z - 1 \leq y < x \leq w\}$. A solution may employ the Floyd-Hoare assignment axiom (1) as well as their proof rule for sequential composition (2) defined below.

$$\{Q[x := e]\}\, x := e\, \{Q\} \quad (1)$$

$$\{P\}S1;\ S2\{Q\} \leftarrow \{P\}S1\{R\} \wedge \{R\}S2\{Q\} \quad (2)$$

An application of (1) replaces all occurrences of x in the postcondition Q with e, thereby obtaining the precondition, indicated by $Q[x := e]$.

In solving the given problem, the assignment axiom is applied by replacing y with $(y - 1)$ in the postcondition Q to obtain an intermediate precondition, say, $P1 = z - 1 \leq (y - 1) < x \leq w$. Thereafter we apply sequential composition and equate $P1$ as the postcondition for the first assignment statement $x := x - 1$. Applying the assignment axiom again in $P1$ gives one the final precondition $P = z - 1 \leq (y - 1) < (x - 1) \leq w$ which can be simplified to $z \leq y < x \leq w + 1$. Therefore: $\{z \leq y < x \leq w+1\}\, x := x-1;\ y := y-1\, \{z-1 \leq y < x \leq w\}$.

An application of the assignment axiom delivers what is known as a weakest precondition, indicated by $wp(S, Q)$ [10] and one may usefully employ this property to verify a given Hoare triple $\{P\}S\{Q\}$ where S may be an arbitrary system. The weakest precondition with respect to S and Q is calculated and a proof obligation $P \Rightarrow wp(S, Q)$ results. If this proof obligation cannot be discharged, chances are that the precondition P suspected beforehand may be too weak, leading to problems further on during development.

The triple $\{P\}S\{Q\}$ is often referred to as a specification for the particular program fragment. The texts by Backhouse [9] or Baber [10] present very detailed accounts of the theory and application of these and other rules.

2.2 Typing

An important extension to the arsenal of verification tools is the notion of *typing*, mainly introduced by Baber [10]. Traditionally the Assignment axiom (e.g. (1) above) is defined without due concern of the types of variables involved. Ignoring types may have severe consequences as is evident in the disaster that struck the European Space Commission's Ariane 5 space rocket during its first test flight on 4 June 1996. A lack of typed verification procedures led to a 64-bit floating point number being assigned to a 16-bit signed integer, resulting in an arithmetic overflow and the initiation of a self-destruct sequence of actions. The code was taken from an earlier rocket model without regard to its precondition. In the earlier model, the precondition was always met, but in the Ariane 5, it was not [12].

Fortunately no human life was lost but the financial loss was in the order of US$500 million [13]. The rocket was furthermore not insured.

It is very plausible that the use of Floyd-Hoare-Baber (FHB) verification principles, especially the use of typing could have prevented the Ariane 5 disaster. This leads to our first observation regarding the use of formal methods at the procedural level of program construction:

- *Observation #1*: The use of FHB techniques may facilitate the correctness of real-life software.

In the next section it is shown how mathematical formalism may be applied to an earlier phase of the Software Development Life Cycle (SDLC) and we argue that it may further facilitate correctness and enhance software development processes.

3 FORMALISING THE SPECIFICATION PHASE

Appendix A illustrates the traditional Software Development Life Cycle (SDLC) and shows how far in the development cycle, namely the coding and implementation phases the pure Floyd-Hoare verification rules were typically used. Notably, these techniques were developed for program constructs only and not the whole of the SDLC activities of that time.

It is quite possible that an error discovered through verification activities during the coding and implementation phases may be traced back to errors during the requirements or specification phases. Early work by Barry Boehm [14] supports this claim. Figure 1 illustrates that discovering and correcting errors later in the life cycle, e.g. during the implementation or integration phases turn out to be more costly than fixing errors during an earlier phase.

Boehm's work was followed more than a decade later by similar case studies regarding the cost of fixing errors progressively later in the SDLC. The later results are shown in Appendix C. This data was gathered, from amongst other projects, during an extensive revamp of a compiler. All errors observed at the end of the design phase were categorised. It is reported that roughly 13% of the faults were inherited from the previous version of the compiler. Of the remaining errors, 16% were introduced during the specification phase and the remaining 71% were introduced during the design phase [15, 16]. These figures call for more attention to be paid to earlier phases since errors tend to snowball through to later phases.

3.1 Embedding FHB in Requirements

The cost effects illustrated by Boehm [14] and Bhandari et *al.* [15], together with *Observation #1* above support the idea of applying formal correctness-preserving techniques during earlier phases. The Floyd-Hoare-Baber approach can usefully be applied to, for example, the specification phase as well (refer Appendix A). In essence a specification of a system defines *what* the resultant system must do rather than saying how it is to be achieved.

The idea of using mathematical formalisms during specification led to the development of a multitude of formal specification languages. Examples of these are VDM [17], RAISE [18], Z [19, 20] and B [21] to name but a few. One of the benefits to be realised through the use of a formal specification is that the specifier may reason *formally* about the properties of the system to be built at a very early stage in the development process [22]. Translating a natural-language statement of requirements into a mathematically formal specification helps one to identify ambiguities, omissions, etc. Any errors or omissions discovered in the user requirements could therefore be rectified earlier, leading to reduced costs.

The specification language Z [19, 20] emerged as a popular choice and enjoyed a fair amount of industrial acceptance (refer Section 10). Z is based on two fundamental concepts in mathematics and computing: set theory and logic. The following section introduces Z and presents some advantages of specification formalisms.

4 FORMAL SPECIFICATION USING Z

Mathematical set theory [23] is a basic, yet deep, underlying commodity in computing. For example, any good text on relational databases shows that databases are based on functions — given a person's

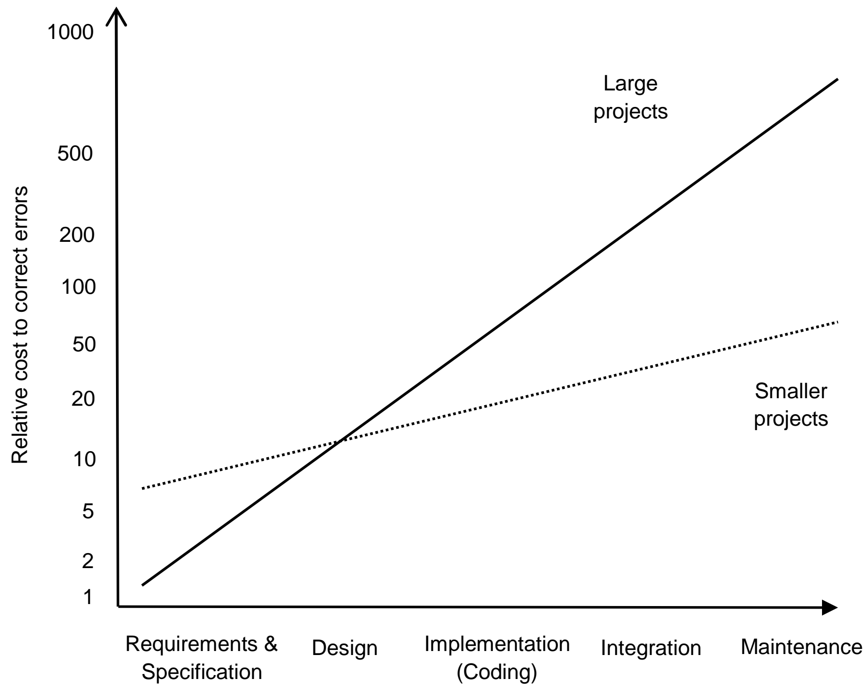


Figure 1: Effect of fixing errors during the life cycle. (Adapted from Boehm [14])

ID number (a key), the system returns the record for the person. It is no coincidence therefore that Z specifications are essentially set-theoretic specifications. In fact, Z is based on a strongly-typed fragment of Zermelo-Fraenkel set theory [23], expressed in suitable first-order languages augmented by *schema notation.* The B system [21] is also based on set theory, mainly because it was was developed from Z. The schema notation arose, according to Hoare, as a mechanism to separate visually the formal parts of a specification from the semi-formal and informal documentation around them [24, 25].

A schema is generally divided into two parts:

- a section in which variables (called components) together with their types are introduced, and
- a predicate in which constraints are placed on the values of the variables.

In the following section a partial Z specification is constructed from a requirements definition. The development is an enhanced version of a system in [26].

4.1 A Specification

Design a Z specification for a system to enable a sea port authority to keep track of oil tankers arriving and docking at its berths. If a tanker arrives, and there is an open berth, it is allocated to the tanker, else the tanker joins the back of a queue, waiting to be berthed. Information to be kept include the berths maintained by the port authority, all tankers known to the authority, a queue of all tankers waiting to be berthed and a record of which tanker occupies which berth.

The following specific requirements are inferred:

(1) A tanker cannot simultaneously be in the queue and in a berth.

(2) The tankers queueing will all be different.

(3) A tanker will queue only if all the berths are full.

(4) The tankers occupying berths will all be different.

(5) Two tankers cannot occupy the same berth.

The first step in constructing a Z specification is to identify the data types that are involved in the spec. These are called the basic types and two such types may be inferred from the above requirements definition:

$[Tanker, Berth]$

Note that it is customary in Z to use singular denotations for sets. Next the state of the system is specified, in essence showing what the system contains and what its prevailing properties are. The state is indicated by $Oil_Tanker_Control$ below.(The numbers in brackets label the formal counterparts of the informal requirements (1) to (5) above).)

$$
\begin{array}{|ll}
\multicolumn{2}{l}{\textit{Oil_tanker_control}} \\
\textit{berths} : \mathbb{P}\,\textit{Berth} & \\
\textit{known} : \mathbb{P}\,\textit{Tanker} & \\
\textit{waiting} : \mathrm{seq}\,\textit{Tanker} & \\
\textit{docked} : \textit{Tanker} \rightarrowtail\!\!\!\!\!+ \textit{Berth} & (4),(5) \\
\hline
\mathrm{ran}\,\textit{waiting} \cap \mathrm{dom}\,\textit{docked} = \varnothing & (1) \\
\#\textit{waiting} = \#(\mathrm{ran}\,\textit{waiting}) & (2) \\
\#\textit{waiting} > 0 \Rightarrow (\textit{berths} = \mathrm{ran}\,\textit{docked}) & (3) \\
\mathrm{ran}\,\textit{docked} \subseteq \textit{berths} & \\
\mathrm{ran}\,\textit{waiting} \cup \mathrm{dom}\,\textit{docked} \subseteq \textit{known} &
\end{array}
$$

Schema *Oil_tanker_control* defines four state components, namely, *berths*, *known*, *waiting* (a sequence) and *docked* (a partial injective function). These represent respectively the set of all berths at the port authority, the set of all tankers that are known to the port authority, the set of all tankers waiting to be berthed and the set of all $(tanker, berth)$ pairs for tankers already berthed. The symbol $\mathbb{P}$ denotes a set-theoretic powerset. The conditions below the short dividing line represent the state invariant.

Observe that every state component (e.g. *berths*) is typed, hence the FHB typing mechanism is captured during the specification phase already. However, Z as a specification tool is different from the Floyd-Hoare-Baber style presented in Section 2, since its notation employs set theory and first-order logic rather than programming constructs. Z deals with preconditions and postconditions but not the programs, since these have not been designed yet. In this sense, Z may be viewed as a "front end" to FHB logic, which is used either to design programs satisfying the Z specification or to verify that given programs satisfy the Z spec.

A successful arrival of a tanker at the port is captured by schema *Arrive*:

$$
\begin{array}{|l}
\textit{Arrive} \\
\Delta\,\textit{Oil_tanker_control} \\
\textit{tanker?} : \textit{Tanker} \\
\textit{report!} : \textit{Report} \\
\hline
(\mathrm{ran}\,\textit{docked} \neq \textit{berths} \land \\
\quad (\exists\, b)(b \in \textit{berths} \land \\
\qquad b \notin \mathrm{ran}\,\textit{docked} \land \\
\qquad \textit{docked}' = \textit{docked} \oplus \{\textit{tanker?} \mapsto b\}) \land \\
\quad \textit{waiting}' = \textit{waiting} \land \\
\quad \textit{known}' = \textit{known} \land \\
\quad \textit{report!} = \textit{OK}) \\
\qquad \lor \\
(\mathrm{ran}\,\textit{docked} = \textit{berths} \land \\
\quad \textit{waiting}' = \textit{waiting} \frown \langle \textit{tanker?} \rangle \land \\
\quad \textit{docked}' = \textit{docked} \land \\
\quad \textit{known}' = \textit{known} \land \\
\quad \textit{report!} = \textit{wait})
\end{array}
$$

The tanker that arrives is denoted by *tanker?*. Δ indicates that, since a tanker arrives at the port there is a possible change in the state. In Z a prime ($'$) denotes the value of a variable after an operation. The symbol $\oplus$ is Z's overriding operator — it replaces tuples in a relation or a function where the first coordinate matches the coordinate of the overriding tuple. Concatenation of sequences is indicated by $\frown$.

The first *disjunct* inside the predicate part specifies what happens when a tanker arrives and there is at least one free berth. The berth is allocated to the tanker. The second disjunct specifies what happens if there is no free berth for the newly arrived tanker — the tanker is added to the back of a waiting queue. Note that for the purposes of this specification, *Arrive* does not specify any error conditions. Schemas defining error conditions may be specified separately and can be combined with the other schemas using the schema calculus [27].

Next we show how an attempt at discharging a proof obligation reveals an omission in the specification.

4.2 The Utility of Proof

Recall that the state of the system is given by *Oil_Tanker_Control* above. Part of the invariant of the *after state* resulting from operation *Arrive* is:

$$\mathrm{ran}\,\textit{waiting}' \cup \mathrm{dom}\,\textit{docked}' \subseteq \textit{known}' \qquad (3)$$

A proof of the above property would confirm that *Arrive* preserves part of the system invariant.

Suppose the first disjunct of *Arrive* holds, i.e., $\mathrm{ran}\,\textit{docked} \neq \textit{berths}$, meaning there is a berth available when a new tanker arrives. Discharging (3) reduces to proving the following:

(1) $\mathrm{ran}\,\textit{waiting}' \subseteq \textit{known}'$

(2) $\mathrm{dom}\,\textit{docked}' \subseteq \textit{known}'$

Case (1) holds, since:

if $\mathrm{ran}\,\textit{waiting} \subseteq \textit{known}$ [before state invariant]
then $\mathrm{ran}\,\textit{waiting}' \subseteq \textit{known}$ [$\textit{waiting}' = \textit{waiting}$]
i.e. $\mathrm{ran}\,\textit{waiting}' \subseteq \textit{known}'$ [$\textit{known}' = \textit{known}$]

Case (2) however reveals a problem:

$$
\begin{aligned}
&\mathrm{dom}\,\textit{docked}' \\
&= \mathrm{dom}\,(\textit{docked} \oplus \{\textit{tanker?} \mapsto b\}) \\
&\qquad\qquad [\exists\, b \in (\textit{berths} - \mathrm{ran}\,\textit{docked})] \\
&= \mathrm{dom}\,\textit{docked} \cup \mathrm{dom}\{\textit{tanker?} \mapsto b\} \qquad [\text{law of } \oplus]
\end{aligned}
$$

Considering the 1st operand of the above union:

$\mathrm{dom}\,\textit{docked} \subseteq \textit{known}$ [invariant before state]

$$\text{i.e. } \mathrm{dom}\ \mathit{docked} \subseteq \mathit{known}' \qquad [\mathit{known}' = \mathit{known}]$$

The last part to prove is:

$$\mathrm{dom}\{\mathit{tanker}? \mapsto b\} \subseteq \mathit{known}'$$
$$\text{i.e. } \{\mathit{tanker}?\} \subseteq \mathit{known}' \qquad [\text{definition of dom}]$$
$$\text{i.e. } \mathit{tanker}? \in \mathit{known}'$$

It follows that for *tanker*? to be an element of known', the precondition of *Arrive* has to be augmented with $\mathit{tanker}? \in \mathit{known}$, i.e. the precondition has to be strengthened.

Further investigation reveals that the precondition needs to be strengthened even more, since it has to provide for the case that a newly arrived tanker must not already be queued or docked. Such details would form part of a more robust version of *Arrive*.

This exercise reveals a very important benefit of a formal specification and a formal-methods approach, leading to our second observation:

- *Observation #2*: A formal-methods approach may facilitate the discovery of errors in the specification, e.g. weak preconditions to operations. This could be vital to FHB verification procedures during later phases of the SDLC.

4.3 Is it Worth the Effort?

The above proof obligation was discharged on paper through sheer human effort. Fortunately there are automated reasoning assistants available to facilitate proofs like the above. In fact, most resolution-based reasoners would successfully discharge the above proof obligation (provided of course the precondition has been sufficiently strengthened). For example, the Otter reasoner written by William C. McCune [28] has little difficulty in finding a proof of the above property.

Much research went into the development of automated or at least semi-automated reasoners over the past couple of decades. In Section 9, an example is presented of a proof obligation that humans tend to have difficulty with, simply because of overwhelming mathematical notation. The proof turns out to be easy, however, for a reasoning assistant. The opposite may also occur. Often a theorem that is easily proven by a trained mathematician is hard for a reasoner. Example 4 in Section 7 presents such a problem.

The following section briefly introduces an early reasoner and it is shown how time consuming it was to prove a simple property. Thereafter some of the advances made over the past couple of decades in the area of automated reasoning are considered. In particular the utility of heuristic reasoning is illustrated.

5 EARLY REASONERS

The paper by Woodrow (Woody) Bledsoe in the collection compiled by Jörg Siekmann and Graham Wrightson [29] gives an account of an early theorem-proving program called PROVER. The prover would divide a problem into subproblems through two routines called SPLIT (for general mathematical problems) and REDUCE (for problems in set theory). The subproblems were then passed on to a resolution procedure to perform the necessary proofs.

The divide-and-conquer approach was very necessary, as Bledsoe wrote: "Resolution, when used, is relegated to the job it does best, proving relatively easy assertions". The capabilities of these early provers were limited, as pointed out again by Bledsoe: "But this ability [dividing a problem into subproblems], which is really an overall *planning* capacity, is still severely limited".

An example of the operation of the split and reduce routines is given below.

Example 2

Suppose a specifier wishes to prove $(x \in (A \cup B) \to P)$ where A, B and P have been defined before. The reduce routine would rewrite the proof obligation as $((x \in A \lor x \in B) \to P)$ whereafter SPLIT would divide it into two subproblems $(x \in A \to P)$ and $(x \in B \to P)$. The resolution procedure would subsequently attempt to refute each of the two subproblems.

If each of the proof attempts $(x \in A \to P)$ and $(x \in B \to P)$ yields a proof, the original conjecture $(x \in (A \cup B) \to P)$ holds. If either of the individual proof attempts fails to produce a proof, the combined conjecture cannot be inferred.

6 ADVANCES MADE BY AUTOMATED REASONERS

Resolution-based reasoners made steady progress since the 1980s. The OTTER (Organised Theorem-proving Techniques for Effective Research) theorem prover [30, 31] is one of the successes in this area. I have used this reasoner extensively (OTTER easily discharges the proof obligation in Example 2 above), and so has one of the founders of Automated Reasoning, Larry Wos [31]. In fact, a variant of OTTER called EQP was used by its author (McCune) to discharge a longstanding open conjecture, namely, that Robbins Algebras are boolean algebras [32]. The automated proof of this famous conjecture made the front pages of national newspapers worldwide [33], leading to our third observation:

- *Observation #3*: Automated reasoning assistants may perform tasks that humans *fail* to achieve.

7 CHALLENGES OF SET-THEORETIC PROOFS

The fact that Z is based on set theory and first-order logic brings about significant challenges when one tries to reason about the properties of such specifications. Set theory is highly hierarchical, since an object (a set) at a very fine level of granularity may be an element of another (coarser) object, which may in turn be a member of another, even coarser object. Iterations through these levels often give rise to much activity and the generation of much irrelevant information [25]. If one, therefore, embarks on the use of an automated reasoner to prove properties of these set-theoretic specifications, then one quickly encounters a number of time- and space complexity problems [34, 35, 25].

A reasoner should avoid 'opening up' every set-theoretic definition so that inferences can be made at the appropriate level. Some definitions must, however, be expanded. A key technique would be to layer the deductions and to identify suitable occasions for crossing from one layer to another. The following example serves as an illustration.

Example 3

Consider the proof obligation:

$$A \subseteq B \rightarrow \mathbb{P}(A) \subseteq \mathbb{P}(B) \tag{4}$$

An automated reasoner attempting to prove (4) should stay at the first level of powersets, e.g. $\mathbb{P}(A)$ and not attempt to move to any 2nd or 3rd levels, e.g. $\mathbb{P}\,\mathbb{P}(A)$ or $\mathbb{P}\,\mathbb{P}\,\mathbb{P}(A)$. Humans readily avoid such pitfalls but an automated reasoner cannot easily make such inferences.

Despite many theoretical advances, set theory pose demanding challenges to automated reasoning programs [34, 35, 25]. These advances include hyperresolution [36], set-of-support strategy [37], paramodulation [38], resonance [39] and the hot-list strategy [40] to name just a few.

There is, however, a further technique that is rather useful, namely, the use of set-theoretic reasoning heuristics discussed below.

7.1 Heuristic Reasoning

It is generally recognised that *heuristics* would play an important role in launching an attack on the complexities of set theory, thereby increasing the success rate of an automated reasoner [33]. An illustration of one of these heuristics from [25] and [22] in reasoning about mathematical set theory is given in Example 4. All the proof attempts reported on in the rest of this paper were done on a 2.2 GHz Dual Core machine with 1GB RAM and a clock speed of 800MHz.

Example 4

Suppose one has to prove:

$$\mathbb{P}\{0,1\} = \{\varnothing, \{0\}, \{1\}, \{0,1\}\} \tag{5}$$

When writing the contents of sets in list notation, e.g. the contents of the above set on the right-hand side, one naturally tends to define these sets using one or more levels of indirection by moving from the various elements to a symbol representing the collection of those elements. Therefore, we rewrite the above set-theoretic equality to make the relevant constructions explicit, i.e.:

$$A = \{0\} \wedge B = \{1\} \wedge C = \{0,1\} \wedge D = \mathbb{P}(C) \wedge \\ E = \{\varnothing, A, B, C\} \rightarrow D = E \tag{6}$$

The OTTER reasoner fails to find any proof for (6) in *30 minutes*. Suppose the complexity of the information is reduced by eliminating the symbol E and listing its contents directly, i.e.

$$A = \{0\} \wedge B = \{1\} \wedge C = \{0,1\} \wedge \\ D = \mathbb{P}(C) \rightarrow D = \{\varnothing, A, B, C\} \tag{7}$$

With the above formulation OTTER finds a proof in *4 minutes 5 seconds*. This brought about an important heuristic proposed in [22], namely, to avoid unnecessary levels of elementhood in set-theoretic formulae by using the elements of sets directly. Through the use of the divide-and-conquer technique, this last proof attempt may be streamlined even further [22]. This example is therefore a case where a human reasoner would easily verify the truth of a formula but a machine experiences difficulty with it.

The following observation emerges from the above proof attempt:

- *Observation #4*: The use of carefully selected heuristics may facilitate automated reasoning, a much needed activity in the process of constructing highly dependable software.

8 NEXT-GENERATION REASONERS

The Vampire theorem prover [41] is currently considered to be the benchmark for resolution-based reasoners, owing to its consistent success at the annual CADE ATP System Competitions (CASC)

[42]. Details of recent CADE competitions appear at http://www.cs.miami.edu/~tptp/CASC/. Vampire also fails to find a proof of (6) in 30 minutes. For format (7), however, it finds a proof in just *0.8 seconds* [43].

The OTTER reasoner has since been decommissioned by McCune and replaced by a more advanced reasoner, Prover9 [28] (see also http://www.cs.unm.edu/~mccune/prover9/). Naturally an important project would be to measure the performance of Prover9 on the set-theoretic proofs in [22].

The following section gives an example of a proof attempt that a human may have difficulty with, but which is easy for an automated reasoner.

9 NOTATIONAL COMPLEXITY

Suppose one has to show:

$$a \in B \to \mathbb{P}(a) \in \mathbb{P}\mathbb{P}\bigcup(B) \tag{8}$$

where $\bigcup$ represents set-theoretic distributive union. For example, $\bigcup\{\{1,2,3\},\{1,4\},\{2,5,6\}\} = \{1,2,3\}\cup\{1,4\}\cup\{2,5,6\} = \{1,2,3,4,5,6\}$. Despite the notational complexity of (8), an automated reasoner (e.g. OTTER) has little difficulty in finding a proof for it. Humans, however, may not find it as easy. This in the opinion of the author portrays the real power of formal methods in computing and may very well lead to some alleviation of the controversy surrounding formal methods, viz (compare with Observation #3 above):

- *Observation #5*: Automated reasoning assistants may perform tasks that humans find *hard*.

In looking through the proof attempts presented in this paper one may be inclined to think that an automated reasoner always finds symbolic proof attempts (e.g. (8)) easy, but experiences difficulty with enumerated (e.g. (5)) proof obligations. This is however not the case, since OTTER fails to find a direct proof of (for example) property (9) below.

$$A \times \bigcup(B) \subseteq \bigcup\{A \times X \mid X \in B\} \tag{9}$$

Some further heuristics are called for. Details appear in [22].

The following sections present some formal methods successes and consider possibilities for improving the chances of formal-methods practices being accepted in industry.

10 SUCCESSES IN INDUSTRY

To date the use of formal methods resulted in a number of industrial successes:

- Arguably the best documented example of the use of Z is CICS (Customer Information Control System) [27]. Through the use of Z a 40% reduction in the number of errors usually found was reported and it was estimated that a huge saving in the total development cost was achieved.
- The Inmos T800 Floating Point Transputer system [44] was specified with Z and it also resulted in cost savings and faster progress towards the end product.
- The Paris Metro System [45] in France is an impressive success story through the use of the B-method [21]. B comprises a complete development environment and has been developed from Z. For this system (supporting conventional and driverless trains) approximately 100,000 lines of B specification was produced. In the order of 28,000 proof obligations were generated. During these proof attempts many errors were found and corrected in the B spec. A subsequent testing process did not find a single error. Details appear in [46].

The following observation may be inferred from the above (compare with Observation #1 in Section 2.2):

- *Observation #6*: The use of formal methods may result in cost savings of industrial software projects.

11 THE FUTURE OF FORMAL METHODS

While the successes of the preceding case studies are encouraging, much work remains before the general use of formal methods may receive industrial acceptance. The following proposals to facilitate the industrialisation of formal methods may be distilled from the literature, e.g. [45]:

(1) *Incremental construction*: It should be possible to build higher-level specifications from provably correct lower-level ones. This principle has generally been well mastered by traditional engineering disciplines, e.g. the construction of a civil engineering artefact.

(2) *Comparative analysis*: It should be possible to quickly construct and compare different specification techniques for a given problem, much like the well-known prototyping paradigm.

(3) *Integration*: It ought to be possible to integrate formal-methods components and techniques into the existing SDLC (refer Appendix A).

(4) *High-level Tool support*: More tool support for the use of formal methods should be developed. These tools should make suggestions, provide

constructive feedback, etc. Reasoning about specifications should be facilitated. The use of heuristics together with an HCI-based library of heuristic patterns should be useful in this regard. Equally, the use of patterns to facilitate the reuse of previously generated proofs should become possible.

(5) *Lightweight techniques*: The learning curve of a formal method should be flattened. It is possible that many software engineers have been turned off mathematics during any of their primary, secondary or tertiary education. Measures to change such attitudes ought be be devised and implemented.

(6) *Partial reasoning*: It should be possible to reason about partially constructed or partially correct specifications, or in spite of errors still present in the specification. An analogy would be executing a program that is not yet fully developed like in an interpreter-based environment.

(7) *Measurable progress*: It must be possible to regularly measure the progress made by a formal technique during development.

12 SUMMARY

An overview of formal software development from the viewpoint of procedural verification techniques and formal specifications was presented. The Floyd-Hoare-Baber contributions to the verification scene during the early years were presented and these were linked with the specification phase of the SDLC. The value of using the FHB techniques in the construction of highly dependable software was emphasised in relation to the Ariane 5 failure.

The importance of identifying and correcting errors early on in development was acknowledged, leading to the use of a formal specification. The integration of a formal, abstract specification with the FHB techniques has been proposed. One of the advantages of using a formal specification is that the software engineer may reason about the correctness of the specification, thereby improving the spec.

Mathematical set theory on which many formal specification languages are based, pose demanding challenges to automated reasoners. The use of heuristics may facilitate the task of discharging proof obligations to increase confidence in the correctness of the specification. Humans and machines have different theorem-proving strengths and may usefully complement each other.

Throughout this paper observations were made on the strength of the material presented. It is hoped that these observations may go some distance in alleviating part of the controversy surrounding the use of formal methods. Examples of formal methods success stories in industry were also presented.

The industrial use of formal methods nevertheless remains a contentious issue and guidelines that may facilitate this process were given.

13 ACKNOWLEDGEMENT

The author would like to thank the reviewers for making valuable suggestions regarding this article.

REFERENCES

[1] J. Woodcock and M. Loomes. *Software Engineering Mathematics: Formal Methods Demystified*. Pitman, London, 1988.

[2] R. W. Floyd. "Assigning Meanings to Programs". *Proceedings of Symposia in Applied Mathematics*, vol. 19, pp. 19 – 32, 1967.

[3] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". *Communications of the ACM*, vol. 12, no. 10, pp. 576 – 580 and 583, October 1969.

[4] J. E. Hopcroft and J. D. Ullman. *Formal Languages and Their Relation to Automata*. Addison-Wesley, Reading, Mass., 1969.

[5] J. D. Ullman. *Principles of Database Systems*. Computer Science Press, 1982.

[6] B. Cohen. "A Rejustification of Formal Notations". *Software Engineering Journal*, vol. 4, no. 1, pp. 36 – 38, January 1989.

[7] B. L. Charlier and P. Flener. "Specifications Are Necessarily Informal or: Some More Myths of Formal Methods". *The Journal of Systems and Software*, vol. 40, no. 3, pp. 275–296, March 1998.

[8] H. M. Deitel and P. J. Deitel. *Java: How to program with an introduction to Visual J++*. Prentice Hall, 1997.

[9] R. Backhouse. *Program Construction: Calculating Implementations from Specifications*. John Wiley and Sons, 2003.

[10] R. L. Baber. *The Spine of Software*. John Wiley and Sons, 1987.

[11] K. Pohl. "The Three Dimensions of Requirements Engineering". In C. Rolland, F. Bodart and C. Cauvet (editors), *Fifth International Conference on Advanced Information Systems Engineering (CAiSE'93)*, pp. 275 – 292. Springer-Verlag, Paris, 1993.

[12] R. L. Baber. "The Ariane 5 explosion as seen by a software engineer", September 2003. http://baber.servehttp.com/Professional/Ariane/Ariane5.htm, Accessed 5 May 2010.

[13] J.-M. Jazequel and B. Meyer. "Design by contract: the lessons of Ariane". *Computer*, vol. 30, no. 1, pp. 129 – 130, January 1997. doi:10.11092.562936.

[14] B. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.

[15] I. Bhandari, M. Halliday, J. Chaar, R. Chillarege, K. Jones, J. Atkinson, C. Lepori-Costello, P. Jasper, E. Tarver, C. Lewis and M. Yonezawa. "In-Process Improvement through Defect Data Interpretation". *IBM Systems Journal*, vol. 33, no. 1, pp. 182 – 214, 1994.

[16] S. Schach. *Object-Oriented and Classical Software Engineering*. McGraw-Hill, 5th edn., 2002.

[17] C. B. Jones. *Systematic Software Development Using VDM*. Prentice Hall International (UK), 1986.

[18] C. George and A. E. Haxthausen. "The Logic of the RAISE Specification Language". In D. Björner and M. Henson (editors), *Logics of Specification Languages*, pp. 349–399. Springer Berlin Heidelberg, 2008.

[19] J. P. Bowen. "Z: A formal specification notation". In M. Frappier and H. Habrias (editors), *Software Specification Methods: An Overview Using a Case Study*, FACIT, chap. 1, pp. 3 – 19. Springer-Verlag, 2001.

[20] D. Lightfoot. *Formal Specification Using Z*. Palgrave Macmillan, 2nd edn., 2001.

[21] J.-R. Abrial. *The **B** Book: Assigning Programs to Meanings*. Cambridge University Press, August 1996. ISBN 0521496195.

[22] J. A. van der Poll. *Automated Support for Set-Theoretic Specifications*. Ph.D. thesis, University of South Africa, June 2000.

[23] H. B. Enderton. *Elements of Set Theory*. Academic Press, Inc., 1977.

[24] C. A. R. Hoare. "Preface". In D. Bjørner, C. Hoare and H. Langmaack (editors), *VDM'90: VDM and **Z** - Formal Methods in Software Development*, no. 428 in LNCS. 1990.

[25] J. A. van der Poll and W. A. Labuschagne. "Heuristics for Resolution-Based Set-Theoretic Proofs". *South African Computer Journal*, vol. 23, pp. 3 – 17, July 1999.

[26] J. B. Wordsworth. *Software Development with **Z***. International Computer Science Series. Addison-Wesley, 1992. A Practical Approach to Formal Methods in Software Engineering.

[27] B. Potter, J. Sinclair and D. Till. *An Introduction to Formal Specification and **Z***. Prentice-Hall, 2nd edn., 1996.

[28] W. W. McCune. "*Prover9 is Better Than Otter*". Argonne Workshop on Automated Reasoning and Deduction (AWARD), August 2005.

[29] W. W. Bledsoe. "Splitting and Reduction Heuristics in Automatic Theorem Proving". In J. Siekmann and G. Wrightson (editors), *Automation of Reasoning: Classical Papers on Computational Logic 1967 to 1970, Volume 2*, pp. 508 – 530. Springer-Verlag, Germany, 1983.

[30] W. W. McCune. *OTTER 3.3 Reference Manual*. Argonne National Laboratory, Argonne, Illinois, August 2003. ANL/MCS-TM-263.

[31] L. Wos. "Milestones for Automated Reasoning with OTTER". *International Journal of Artificial Intelligence*, vol. 15, no. 1, pp. 3 – 19, February 2006.

[32] W. W. McCune. "Solution of the Robbins Problem". *Journal of Automated Reasoning*, vol. 19, no. 3, pp. 263 – 276, 1997.

[33] A. Bundy. "A Survey of Automated Deduction". Tech. Rep. EDI-INF-RR-0001, Division of Informatics, University of Edinburgh, April 1999.

[34] R. Boyer, E. Lusk, W. McCune, R. Overbeek, M. Stickel and L. Wos. "Set Theory in First-Order Logic: Clauses for Gödel's Axioms". *Journal of Automated Reasoning*, vol. 2, no. 3, pp. 287 – 327, September 1986.

[35] A. Quaife. *Automated Development of Fundamental Mathematical Theories*. Automated Reasoning Series. Kluwer Academic Publishers, 1992.

[36] J. A. Robinson. "Automatic Deduction with Hyper-Resolution". *International Journal of Computer Mathematics*, vol. 1, pp. 227 – 234, 1965.

[37] L. Wos, D. Carson and G. Robinson. "Efficiency and Completeness of the Set of Support Strategy in Theorem Proving". *Journal of the Association for Computing Machinery*, vol. 12, no. 4, pp. 536 – 541, October 1965.

[38] G. Robinson and L. Wos. "Paramodulation and Theorem Proving in First-Order Theories with Equality". In B. Meltzer and D. Michie (editors), *Machine Intelligence, Volume 4*, pp. 135 – 150. Edinburgh University Press, Edinburgh, 1969.

[39] L. Wos. "The Resonance Strategy". *Computers and Mathematics with Applications*, vol. 29, no. 2, pp. 133 – 178, February 1995. (Special issue on Automated Reasoning).

[40] L. Wos and G. W. Pieper. "The Hot List Strategy". *Journal of Automated Reasoning*, vol. 22, no. 1, pp. 1 – 44, 1999.

[41] A. Voronkov. "The Anatomy of Vampire: Implementing Bottom-Up Procedures with Code Trees". *Journal of Automated Reasoning*, vol. 15, no. 2, pp. 237 – 265, 1995.

[42] F. J. Pelletier, G. Sutcliffe and C. Suttner. "The Development of CASC". *AI Communications*, vol. 15, no. 2, pp. 79 – 90, 2002.

[43] P. S. Steyn and J. A. van der Poll. "Validating Reasoning Heuristics Using Next-Generation Theorem Provers". In *The 5th International Workshop on Modelling, Simulation, Verification and Validation of Enterprise Information Systems (MSVVEIS)*, pp. 43 – 52. Funchal, Madeira, Portugal, June 2007. ISBN 978-972-8865-95-5.

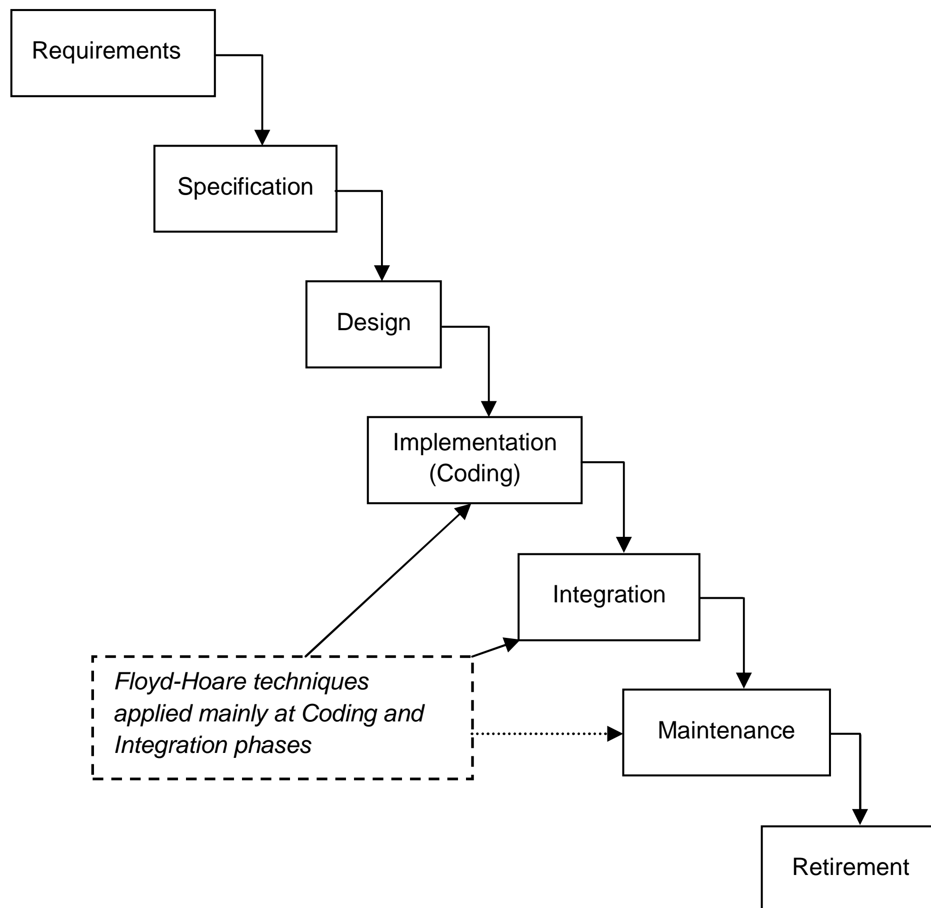
[44] INMOS. *Specification of Instruction Set / Specification of Floating Point Instructions*. Prentice Hall, 1988. INMOS Ltd.

[45] A. van Lamsweerde. "Formal Specification: A Roadmap". In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pp. 147–159. ACM, New York, NY, USA, 2000. ISBN 1-58113-253-0. doi: http://doi.acm.org/10.1145/336512.336546.

[46] P. Behm, P. Benoit, A. Faivre and J.-M. Meynadier. "Météor: A Successful Application of B in a Large Project". In *FM '99: Proceedings of the World Congress on Formal Methods in the Development of Computing Systems-Volume I*, pp. 369–387. Springer-Verlag, London, UK, 1999. ISBN 3-540-66587-0.

[47] B. Hughes and M. Cotterell. *Software Project Management*. McGraw-Hill, 5th edn., 2009.

APPENDIX A: A TRADITIONAL SOFTWARE DEVELOPMENT LIFE CYCLE



The traditional, waterfall-like SDLC [16] is shown above. The ISO 12207 model contains a number of additional processes, e.g. Architectural design, Qualification tests, etc. Details may be found in [47]. The pure Floyd-Hoare verification techniques were applied closer to the end of the cycle. In this paper it is suggested that the full FHB techniques be applied earlier in the cycle.

APPENDIX B: FLOYD-HOARE-BABER VERIFICATION RULES

Skip Statement

$$\frac{}{\{P\}\, skip\, \{P\}}$$

Assignment Axiom

$$\frac{}{\{P[x := E]\}\, x := E\, \{P\}}$$

Sequential Composition

$$\frac{\{P\}\, S\, \{Q\}, \{Q\}\, T\, \{Q\}}{\{P\}\, S, T\, \{R\}}$$

Conditional Statement

$$\frac{\{B \wedge P\}\, S\, \{Q\}, \{\neg B \wedge P\}\, T\, \{Q\}}{\{P\}\, \text{if}\, B\, \text{then}\, S\, \text{else}\, T\, \text{endif}\, \{Q\}}$$

Consequence Rule

$$\frac{P \Rightarrow P', \{P'\}\, S\, \{Q'\}, Q' \Rightarrow Q}{\{P\}\, S\, \{Q\}}$$

While Statement

$$\frac{\{P \wedge B\}\, S\, \{P\}}{\{P\}\, \text{while}\, B\, \text{do}\, S\, \{\neg B \wedge P\}}$$

APPENDIX C - COST OF CORRECTING ERRORS DURING THE SDLC

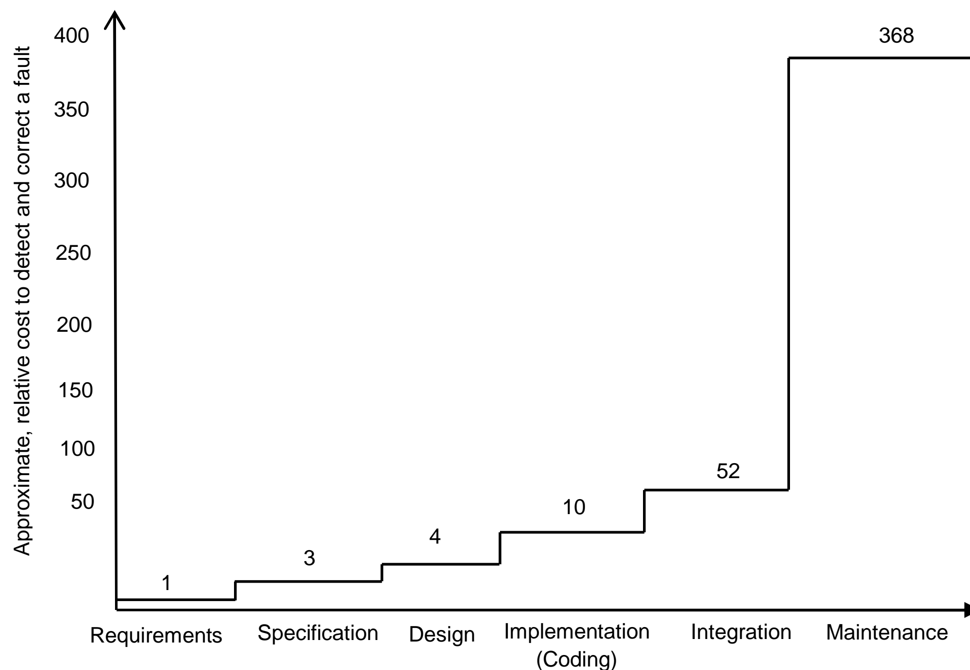


Figure 2: Cost of fixing errors during the SDLC (Bhandari et al. [15])

Later results on the effect of rectifying errors during the SDLC are depicted above. A similar trend as before is observed, namely, the correction of errors as development progresses becomes increasingly more expensive.